Componentes Enterprise JavaBeans

Sesión 4 - Beans asíncronos, temporizadores e interceptores

Acceso asíncrono

- La especificación 3.1 permite el acceso asíncrono a los beans
- Se define con la anotación **`@Asynchronous`** a nivel de clase o de método
- Si el método es **`void`**, el cliente usa el patrón *fire-and-forget*
- Si el método devuelve un valor, debe ser del tipo **`Future<V>`** (interfaz del API de concurrencia de Java)
- El contenedor devuelve el objeto **`Future<V>`** en la invocación inicial, sobre el que el cliente puede consultar el estado de la invocación
- El constructor de la clase **`AsyncResult`** devuelve el valor resultante cuando termina el método

```
@Stateless
@Asynchronous
public class MyAsyncBean {
    public Future<Integer> addNumbers(int n1, int n2) {
        Integer result;
        result = n1 + n2;
        // simulamos una consulta muy larga ...
        return new AsyncResult(result);
    }
}
```

Interfaz Future<V>

- `boolean cancel(boolean mayInterruptIfRunning)`:
  Intenta cancelar la ejecución de la tarea.
- `V get()`:
  Espera si es necesario a que la computación se complete y luego devuelve el resultado.
- `V get(long timeout, TimeUnit unit)`:
  Igual que el método anterior, pero con un timeout.
- `boolean isCancelled()`:
  Devuelve **`true`** si la tarea ha sido cancelada antes de ser completada normalmente.
- `boolean isDone()`:
  Devuelve **`true`** si la tarea ha terminado.

```
@EJB MyAsyncBean asyncBean;

Future<Integer> future = asyncBean.addNumbers(10, 20);
while (true) {
   if (future.isDone()) {
      resultado = future.get();
      break;
   }
   // hago otras cosas que pueda ir adelantando
}
```

Temporizador

- Los schedulers (planificadores) permiten realizar tareas periódicas y asociadas a momentos precisos del tiempo
- Comunes en sistemas operativos (cron) y en aplicaciones de negocios
- Necesarias aplicaciones externas
- El servicio de temporizador (Timer) de Java EE es un primer intento de estandarización y de definición de estas funciones en la propia aplicación

Funcionamiento de temporizadores

- Un bean utiliza el servicio de temporizador (`TimerService`) para obtener por inyección de dependencias un servicio de temporizadores asociado al bean
- Se crea un temporizador con llamadas a `createTimer`
- Los métodos de negocio que nos interesen planificar se anota con la etiqueta `@Timeout`, cuando un temporizador termina, se lanzan todos los métodos con esa anotación

Ejemplo de creación

```java
@Stateless
public class TimerSessionBean implements TimerSessionLocal {
    @Resource
    TimerService timerService;

    public void setTimer(long intervalDuration) {
        Timer timer = timerService.createTimer(
                    intervalDuration,
                    "Creado un nuevo timer");
    }

    @Timeout
    public void timeout(Timer timer) {
        System.out.println("Se ha lanzado el timeout");
    }
}
```

```java
Calendar unoDeMayo = Calendar.getInstance();
unoDeMayo.set(2008, Calendar.MAY, 1, 12, 0);
long tresDiasEnMilisecs = 1000 * 60 * 60 * 24 * 3;
timerService.createTimer(unoDeMayo.getTime(),
                         tresDiasEnMilisecs,
                         "Mi temporizador");
```

Tipos de temporizadores y parámetros

- Dos tipos de temporizadores: de parada única y periódica

Type of Timer	`createTimer` with Parameters
Single-event timer	`createTimer(long timeoutDuration, Serializable info)`
Single event with expiration date	`createTimer(Date firstDate, Serializable info)`
Interval timer with initial expiration	`createTimer(Date firstDate, long timeoutInterval, Serializable info)` or `createTimer(long timeoutDuration, long timeoutInterval, Serializable info)`

Interfaz TimerService

Method Summary	
Timer	**createTimer**(Date initialExpiration, long intervalDuration, Serializable info) Create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval.
Timer	**createTimer**(Date expiration, Serializable info) Create a single-action timer that expires at a given point in time.
Timer	**createTimer**(long initialDuration, long intervalDuration, Serializable info) Create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval.
Timer	**createTimer**(long duration, Serializable info) Create a single-action timer that expires after a specified duration.
Collection	**getTimers**() Get all the active timers associated with this bean.

Interfaz Timer

Method Summary	
void	**cancel**() Cause the timer and all its associated expiration notifications to be cancelled.
TimerHandle	**getHandle**() Get a serializable handle to the timer.
Serializable	**getInfo**() Get the information associated with the timer at the time of creation.
Date	**getNextTimeout**() Get the point in time at which the next timer expiration is scheduled to occur.
long	**getTimeRemaining**() Get the number of milliseconds that will elapse before the next scheduled timer expiration.

Características adicionales

- Son serializables
  - Podemos grabar un temporizador serializando su `TimerHandler`
- Son persistentes
  - Sobreviven caídas y paradas del sistema
- Inconvenientes
  - No son tan flexibles como el cron de Unix
  - No se pueden utilizar en clases Java normales, requieren el uso de EJBs de sesión

Interceptores - Motivación

- Supongamos que queremos calcular el tiempo que tarda en ejecutarse un método:

```
public void addMensajeAutor(String nombre, String texto) {
    long startTime = System.currentTimeMillis();
    Autor autor = findAutor(nombre);
    if (autor == null) {
        autor = new Autor();
        autor.setNombre(nombre);
        em.persist(autor);
    }
    mensajeService.addMensaje(texto, nombre);
    long endTime = System.currentTimeMillis() - startTime;
    System.out.println("addMensajeAutor() ha tardado: " +
            endTime + " (ms)");*
}
```

Problemas del ejemplo anterior

- Se ha añadido en el método `addMensajeAutor()` código que no tiene nada que ver con la lógica de negocio de la aplicación. El código se ha hecho más complicado de leer y de mantener.
- El código de análisis no se puede activar y desactivar a conveniencia. Hay que comentarlo y volver a recompilar la aplicación.
- El código de análisis es una plantilla que podría reutilizarse en muchos métodos de la aplicación. Pero escrito de esta forma habría que escribirlo en todos los métodos en los que queramos aplicarlo.

Solución: interceptores

- Definen código que se puede colocar entre la llamada del cliente a un método y la invocación del método en el bean
- El código del interceptor “envuelve” el método del EJB completamente, capturando las posibles excepciones que éste genera
- Podemos analizar en el código los parámetros proporcionados por la llamada del cliente
- AOP: Programación Orientada a Aspectos

Ejemplo de un interceptor

```
@Interceptors(Profiler.class)
   public void addMensajeAutor(String nombre, String texto) {
      // ...
   }
```

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class Profiler {

   @AroundInvoke
   public Object profile(InvocationContext invocation)
            throws Exception {
      long startTime = System.currentTimeMillis();
      try {
         return invocation.proceed();
      } finally {
         long endTime = System.currentTimeMillis() - startTime;
         System.out.println("El método " + invocation.getMethod() +
                  " ha tardado " + endTime + " (ms)");
      }
   }
}
```

Interceptor que se aplica a todos los métodos de un bean

```java
@Stateless
@Interceptors(Profiler.class)
public class AutorServiceBean implements AutorServiceLocal {

    @PersistenceContext(unitName = "ejb-jpa-ejbPU")
    EntityManager em;
    @EJB
    MensajeServiceDetachedLocal mensajeService;

    public Autor findAutor(String nombre) {
        return em.find(Autor.class, nombre);
    }
    // ...
```

¿Preguntas?